CI1056 - Algoritmos e Estruturas de Dados II

André Grégio e Marcos Castilho
Departamento de Informática – UFPR, Curitiba/PR

Segundo exercício prático

---

1 Introdução

Ordenar uma base de dados é um problema clássico em Ciência da Computação. Muitos pesquisadores têm investido longo tempo em pesquisas para aperfeiçoar as técnicas que aumentam o desempenho dos algoritmos que ordenam dados.

A título de exemplo, o *quick sort*, que é provavelmente o algoritmo de ordenação mais famoso e mais estudado em todos os tempos, ainda é objeto de estudos sérios.

Do ponto de vista da computação teórica, um problema pode ser resolvido por diversos algoritmos diferentes. Os teóricos analisam cada um desses conjuntos de algoritmos como sendo uma *classe de algoritmos*.

Os algoritmos, teoricamente, são analisados segundo os critérios de melhor caso, pior caso e caso médio. É sabido que a análise do caso médio é sempre mais complexa e, em geral, envolve um pouco de conhecimentos probabilísticos (notadamente, sobre distribuição de probabilidades).

Em nossos estudos sobre algoritmos de ordenação, inicialmente vimos os mais simples, que resultaram em algoritmos que apresentam um comportamento quadrático com relação ao tamanho dos dados de entrada.

Os algoritmos *selection sort*, *bubble sort* e *insertion sort* foram analisados formalmente tanto do ponto de vista da complexidade quanto ao número de comparações feitas sobre os dados, quanto do ponto de vista do número de trocas que devem ser feitas para se obter os dados ordenados.

Logo, desses estudos, sabemos que estes três algoritmos requerem um total, no pior caso, de um número quadrático de comparações, e que eles variam com relação ao número total de trocas de elementos. Para estes três algoritmos, vimos que todos requerem algo proporcional ao quadrado da quantidade de dados para comparar os elementos e os ordenar.

Além disso, o número de trocas de elementos varia bastante: alguns fazem uma quantidade proporcional ao tamanho do conjunto de dados, enquanto outros podem fazer até mesmo o quadrado disso.

Vimos também que existem algumas situações especiais que um algoritmo pode ser muito mais eficiente do que outro, e que dependem de como os dados de entrada são fornecidos. A título de exemplo, citamos que, quando os dados são fornecidos já em ordem, o *insertion sort* tem custo linear, enquanto que o *selection sort* mantém seu custo quadrático independentemente dos dados de entrada estarem ordenados. Já o *bubble sort*, em versões otimizadas, pode tratar com eficiência casos particulares de entradas ordenadas.

Em todo caso, todos estes três algoritmos têm custo quadrático no caso médio, embora empiricamente se tem alguma evidência de que o *insertion sort* seja mais eficiente.

Sobre este último comentário, as análises experimentais sempre nos dão evidências de que isto de fato ocorre. Por outro lado, as análises teóricas são extremamente complicadas para se provar matematicamente que isto de fato é verdade.

Enfim, temos um problema comum na física que divide teóricos e experimentais.

2 Os algoritmos da classe $nlog(n)$

Nós estudamos recentemente dois algoritmos que, teoricamente, fato comprovado, fazem a ordenação de uma determinada entrada de dados usando um número de comparações proporcional a $nlog(n)$, sendo n o tamanho da entrada.

Os algoritmos estudados foram o *quick sort* e o *merge sort*, ambos apresentando custo $nlog(n)$ no melhor caso. O pior caso favorece o *merge sort*, que mantém seu custo $nlog(n)$ em detrimento do *quick sort* que degrada para um custo quadrático.

No caso médio, conforme vimos (não totalmente ainda) os dois algoritmos mantém o custo $nlog(n)$.

Uma outra discussão interessante entre os pesquisadores destes algoritmos é uma certa tendência a se acreditar que o *quick sort* é o mais rápido algoritmo que, na média, ordena os elementos de entrada.

De fato, o *quick sort* é, como já dito, provavelmente o algoritmo mais estudado de todos os tempos. Por outro lado, alguns pesquisadores ainda são céticos e continuam a tentar provar que o *merge sort*, ou outros que ainda não estudamos, são no mínimo tão bons quanto o *quick sort*.

Enfim, existe sempre o problema entre teoria e prática...

Como saber ao certo qual algoritmo é o melhor? Existem evidentemente duas abordagens: a primeira é a teórica, a outra é a experimental. As análises teóricas estão sendo feitas em sala de aula, e como este é um trabalho prático, nós vamos tentar fazer análises empíricas que podem nos dizer algo sobre alguns aspectos destes dois algoritmos ditos eficientes.

3 O trabalho

Os teóricos nos dão algumas conjecturas que queremos comprovar através de experimentos:

- O *quick sort* é mais rápido, em tempo de relógio, do que o *merge sort*;
- O *merge sort* tem três otimizações, discutidas em sala de aula, que teoricamente o tornam mais eficiente:
  - Alternar entre o vetor auxiliar e o vetor de entrada nas chamadas recursivas;
  - Testar se o vetor já está ordenado antes de chamar a intercalação;
  - Passar o vetor auxiliar como parâmetro e não como variável local.
  - O *quick sort* também tem algumas técnicas que, teoricamente, o tornam mais eficiente;
    - Tentar outras estratégias de escolha do pivô; a principal delas é a "mediana de três" (ou de cinco?);
    - Quando o sub arranjo tiver um tamanho pequeno chamar o *insertion sort* em vez de realizar as chamadas recursivas do *quick sort*. Alguns autores dizem que "pequeno" significa 15, outros 30. Qual será o valor ideal para os computadores de hoje?
    - Pesquisas também indicam que, antes de se chamar o *quick sort*, embaralhar os elementos ajuda bastante, o que não é muito intuitivo pois se gasta tempo embaralhando os elementos. Em teoria, isto melhora a eficiência do algoritmo, conforme veremos, mas, e na prática? Qual o gasto adicional de embaralhamento para vetores muito grandes?
    - O *quick sort*, teoricamente, é sensível à maneira como os dados estão apresentados, podendo levar a um custo quadrático. Isto se confirma na prática? Para qual tipo de vetores de entrada?

4 Afinal, qual é o trabalho a ser feito?

Neste trabalho você irá analisar todas as variantes citadas acima de maneira empírica. Para isso, você receberá um arquivo de *headers* (`lib_aula19.h`) com os protótipos que você deverá usar, e claro, implementar em um arquivo `lib_aula19.c`. Depois implemente um programa `analise.c` que use esta biblioteca para programar o que está especificado abaixo. Um arquivo `analise.c` é fornecido como um exemplo do que você deve fazer, em particular quanto ao uso da biblioteca `time.c`.

Com base nestes protótipos, produza gráficos usando o *gnuplot*, que você já aprendeu a usar, que procurem dar evidências das seguintes suposições:

1. Sobre o *quick sort*, implementação básica, contra o *merge sort*, considerando que o *quick sort* usa o pivô sendo o primeiro elemento do conjunto, qual algoritmo é mais rápido:

   - No caso de entradas com elementos aleatoriamente distribuídos;
   - No caso de entradas contendo elementos distribuídos segundo o pior caso.

2. Encontre um tamanho de entrada para o qual o *insertion sort* é mais eficiente que o *quick sort*. Idem para o *merge sort*

3. No caso particular do *quick sort*, verifique se a técnica de escolher o pivô no início é melhor do que a média de três ou não. No caso da mediana de três, escolha quaisquer três elementos e o pivô será a mediana destes três;

4. Será que a mediana de cinco é melhor do que a mediana de três? Comprove empiricamente;

5. No caso particular do *merge sort*, comprove empiricamente que as três potenciais melhorias citadas acima de fato melhoram o comportamento dele;

6. Finalmente, dado que você já calculou o valor tal que o *insertion sort* é melhor do que o *quick sort* implemente a seguinte ideia: Quando a recursão chegar em um tamanho menor ou igual a este valor, no lugar de chamar o *quick sort* recursivamente, chame o *insertion sort*.

   - Analise o impacto para um vetor de entrada qualquer;
   - Analise o impacto para um vetor de entrada do tipo "pior caso". enditemize

Para você poder fazer este trabalho você deverá trabalhar com tamanhos de entrada diversos. Inicie com vetores de entrada de tamanho 10 e vá aumentando a ordem de grandeza, isto é, multiplicando por 10 estes tamanhos, até o limite de 1 milhão. Isto é, entradas de tamanho 10, 100, 10000, 100000, 1000000. Na verdade, você pode parar na ordem de grandeza a partir da qual seu computador dá erro de alocação. A título de exemplo, no meu computador eu consegui ir até 100 milhões, mas não 1 bilhão. Lembre que algoritmos recursivos consomem a pilha de alocação...

Ao final, você deve gerar gráficos que usam o gnuplot para cada uma das diversas análises feitas comparando o tempo do algoritmos de ordenação aqui mencionados.